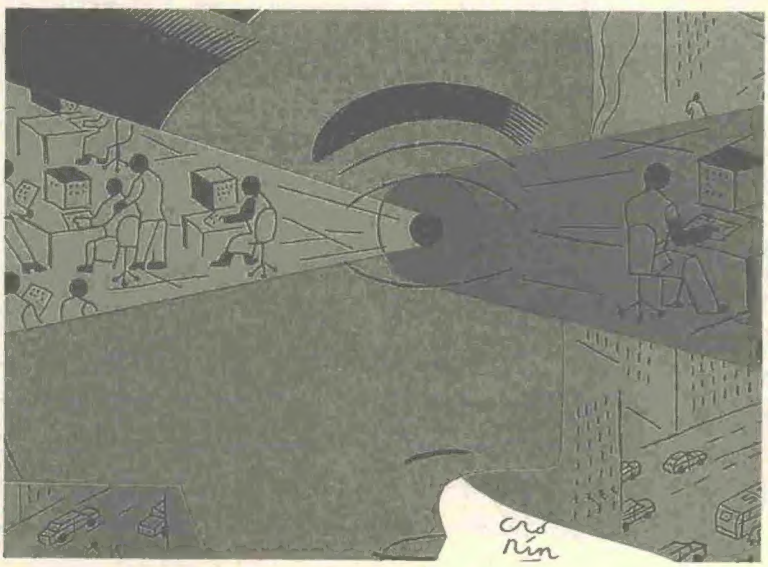
Delphi

ประพนธ์ อัศวภาณุวัฒน์

Multitasking & Multi-Threading

SOFT MAIL

แนวความคิดในการทำงาน
ของระบบ Thread Objects
และเพื่อที่จะให้อยู่ในรูปแบบ
การเขียนโปรแกรมแบบ Objects
จึงใช้ Delphi's TThread Objects Class
เป็นตัวอย่างหลัก



มัลติทาสกิง (Multitasking) เป็นคำอธิบายการทำงานแบบหลาย ๆ แอพพลิเคชัน (Applications) พร้อม ๆ กันบนระบบเดียวกัน ระบบปฏิบัติการหรือโอเอส (Operating System) บน PC ในอดีตถึงปัจจุบันต่างก็พยายามผลักดันให้ตนเองมีความสามารถรันโปรแกรมหรือแอพพลิเคชันพร้อม ๆ กันได้ในลักษณะ Parallel Running คือสามารถทำงานไปพร้อม ๆ กันได้โดยไม่ต้องหยุดโปรแกรมตัวอื่นเลย สำหรับระบบที่มีความนิยมมากที่สุดและที่จะกล่าวถึงก็คือระบบไมโครซอฟท์วินโดวส์

Microsoft Windows 3.x

ไมโครซอฟท์วินโดวส์ รุ่น 3.x เป็นระบบปฏิบัติการ 16 บิตมัลติทาสกิง ชนิด *Non-preemptive* หรือ *Task Switching* กล่าวคือ ไม่ได้เป็นระบบมัลติทาสกิงอย่างแท้จริง เพียงแต่แบ่งช่วงเวลา (Time Slice) ให้แอพพลิเคชันหนึ่งสามารถประมวลผลได้ด้วย Control ทั้งหมด และรอจนเมื่อได้รับ Control กลับมาแล้วจึงสลับให้แอพพลิเคชันอื่นประมวลผลต่อไป ทำให้ดูเสมือนแอพพลิเคชันต่าง ๆ สามารถทำงานไปพร้อม ๆ กันได้

เราสามารถทดสอบการทำงานของระบบทาสก์ (Task) ชนิด Non-preemptive บน Windows 3.x ได้โดยสร้างโพรซีเดอร์ ดังนี้

```
Procedure Tform.LoopForEver;
Begin
   While ( True ) do Begin  End;
End;
```

การทำงานของกิจกรรมข้างต้นจะเป็นการยึด Control ไว้ทั้งหมด โดยไม่คืนให้กับระบบ เมื่อเรียกใช้กิจกรรมนี้แล้ว โปรแกรมจะเข้าสู่วนรอบไม่รู้จบ ทำให้ไม่สามารถส่ง Control คืนให้กับระบบวินโดวส์ เป็นผลทำให้ไม่สามารถควบคุมระบบวินโดวส์ได้อีกต่อไป วิธีที่จะยุติการรันทำได้เพียงอย่างเดียวคือ Reset เครื่องใหม่เท่านั้น

ข้อควรระวัง : ก่อนรันกิจกรรมนี้ควร Save และปิดแอพพลิเคชันที่ใช้งานอยู่ทุกตัว และกิจกรรมนี้จะสามารถใช้ทดสอบได้บนระบบวินโดวส์รุ่น 3.x เท่านั้น

เหตุที่เป็นเช่นนั้นเนื่องมาจากการจัดทาสก์ของระบบ Non-preemptive บนวินโดวส์ 3.x ไม่ได้แบ่งตาม System Timer แต่เป็นการอนุญาตให้แอพพลิเคชันครอบครองการทำงานของ CPU ทั้งหมดในช่วงเวลาหนึ่ง ๆ (Time Slice) และถ้าหากแอพพลิชันไม่คืนการทำงานให้วินโดวส์ดังเช่นตัวอย่างโปรแกรมข้างต้น ระบบวินโดวส์ก็จะไม่สามารถดำเนินการต่อไปได้

อาจจะกล่าวได้ว่าระบบวินโดวส์ 3.x ยังไม่ล้มหายตายจากไปอย่างสิ้นเชิง เพราะเนื่องจากเป็นระบบที่ค่อนข้างมีประสิทธิภาพ และมีความต้องการทางด้านฮาร์ดแวร์ค่อนข้างต่ำ คือเพียงเครื่องระดับ 386 พร้อม RAM 4 MB ก็สามารถรันโปรแกรมต่าง ๆ ได้ ดังนั้นการพัฒนาโปรแกรมบนวินโดวส์ 3.x ผู้พัฒนาโปรแกรมส่วนใหญ่จะต้องระมัดระวังเรื่องลอจิกในส่วนของคำสั่งวนรอบต่าง ๆ เพื่อป้องกันการเกิดการวนรอบไม่รู้จบ การแก้ไขปัญหาของผู้ที่จะต้องพัฒนาโปรแกรมบนระบบวินโดวส์ 3.x อาจทำได้โดยการเตรียมกิจกรรมควบคุมการทำงานโดยควบคุมการไหลเวียนของระบบเมสเสจ (ระบบการทำงานของเมสเสจสามารถอ่านได้ในเล่ม 145-146 ครับ) เราสามารถเขียนกิจกรรมควบคุมได้ดังนี้ครับ

```
Procedure Tform.Schedule;
Var Msg : TMsg;
Begin
  While PeekMessage(Msg,0,0,0,PM_REMOVE) do
  begin
    TranslateMessage(Msg);
    DispatchMessage(Msg);
  end;
End;
```

กิจกรรมนี้เป็นกิจกรรมที่ทำหน้าที่ตรวจสอบและประมวลผลเมสเสจ ถ้าเราใส่กิจกรรมนี้ในกิจกรรม LoopForEver ข้างต้น เป็น

```
Procedure Tform.LoopForEver;
Begin
  While ( True ) do Begin Schedule; End;
End;
```

ผลที่ได้คือ ถึงแม้ Applications (form) นี้อยู่ในสถานะวนรอบไม่รู้จบ ระบบวินโดวส์ยังคงสามารถทำงานต่อไปได้ โดยเรายังคงสามารถยุติการรันวินโดวส์ได้ เนื่องจากในส่วนการวนรอบไม่รู้จบทุกครั้งที่มีการวนรอบได้ทำการเรียกกิจกรรม Schedule ซึ่งเป็นกิจกรรมที่ทำหน้าที่ถอนและแปลเมสเสจที่ค้างอยู่ใน Queue โดยส่งให้กับระบบวินโดวส์จัดการ ทำให้ระบบไม่อยู่ในสถานะ Dead lock จึงสามารถดำเนินการอย่างอื่นต่อไปได้

Microsoft Windows 95, Microsoft Windows NT

ไมโครซอฟท์วินโดวส์ 95 และไมโครซอฟท์วินโดวส์ NT เป็นระบบปฏิบัติการแบบ 32 บิตมัลติทาสกิง ชนิด Preemptive (จะไม่ขอกล่าวถึงความแตกต่างระหว่าง Windows 95 และ Windows NT ในที่นี้) กล่าวคือ ภายใต้ระบบ Win32 นี้ ระบบจะทำการสร้างหน่วยความจำเสมือน (Virtual Address Space) รวมทั้งแบ่งเวลาการประมวลผลให้ Process ต่าง ๆ โดย Process แต่ละตัวมีหน่วยความจำและเวลาในการประมวลผลเป็นอิสระต่อกัน

Process คือแอพพลิเคชันหนึ่ง ๆ ที่สร้างขึ้นเพื่อจุดประสงค์ใด ๆ Process จะมีส่วนที่เป็นข้อมูล (data) รีซอร์ส (resource) และหน่วยความจำเสมือน (ขนาด 2 GB) เป็นของตนเอง ซึ่งถูกใช้โดย "เทร็ด" (Threads) ที่ทำงานอยู่ภายใน Process เอง

Threads คือ Win32 ออบเจ็กต์ (Objects) ที่รัน (Executing) อยู่ภายใน Process เมื่อระบบ Win32 ทำการสร้าง Process ขึ้นในครั้งแรก (เมื่อเราโหลดโปรแกรมที่ต้องการใช้งานเข้าหน่วยความจำ) Thread สำหรับ Process นี้ก็จะถูกสร้างขึ้นพร้อม ๆ กันเพื่อรันโค้ดให้ได้ผลลัพธ์ตามต้องการ เราเรียก Thread นี้ว่า "*Primary thread*" โดยปกติแล้วใน Process หนึ่ง ๆ สามารถมี Threads ได้มากกว่า 1 ตัว โดยแต่ละตัวจะรันอยู่ภายใต้หน่วยความจำเสมือนเดียวกัน และ Thread แต่ละตัวสามารถมีโค้ดที่แตกต่างกันได้ Process ใด ๆ ที่มี Thread รันอยู่มากกว่า 1 มักเรียกว่า "*Multi-Threaded Applications*"

ระบบ Multi-Threaded Process มักนำมาใช้ในงานประมวลผลประเภท Background เป็นส่วนใหญ่ เช่น คำนวณผลลัพธ์ใน Cells ของโปรแกรมประเภท Spreadsheet ทำ Spell checking ของโปรแกรมประเภท Word Processor หรือ Spooling process สั่งพิมพ์เอกสารไปยังพรินเตอร์ เป็นต้น

Preemptive vs Nonpreemptive Multitasking

ถึงแม้ Win32 จะเรียกได้ว่าเป็น True Multitasking แต่ก็ไม่ได้หมายความว่าจะสามารถรัน Thread Objects หลาย ๆ ตัวพร้อมกันได้ในช่วงเวลาเดียวกัน (จนกว่าระบบคอมพิวเตอร์ที่ใช้จะมี CPU เท่ากับจำนวน Thread Objects ที่รันอยู่) แต่ Win32 ใช้เทคนิคการแบ่ง Time Slice ให้ Thread Objects แบบ Quantum คือแบ่งให้ในปริมาณที่เท่ากัน โดย Thread Objects แต่ละตัวจะมีส่วนที่เรียกว่า "คอนเท็กซ์" (Context) ใช้เก็บข้อมูลสำคัญต่าง ๆ เช่น ค่าของรีจิสเตอร์ที่ใช้งาน เพื่อที่จะสามารถทำกิจกรรมของตนแบบต่อเนื่องได้ เงื่อนไขสำคัญของการทำงานในระบบ Win32 ที่ทำให้ Thread Objects แต่ละตัวมีกิจกรรมที่ทำให้ไม่ดึงวงรอบ CPU (CPU Cycles) ไว้กับตนเองทั้งหมด คือ ***Thread Objects แต่ละตัวจะตื่นขึ้นมาทำกิจกรรมของตน ในระหว่างช่วงเวลาของตนเท่านั้น*** (ถ้าเปรียบเทียบวงรอบ CPU เป็นเข็มนาฬิกา Thread Objects ก็คือตำแหน่งเวลาต่าง ๆ ที่อยู่ในนาฬิกาเรือนนั้น เมื่อเข็มนาฬิกาชี้ไปที่ตำแหน่งใด ๆ Thread Objects ที่ตำแหน่งนั้นก็จะตื่นขึ้น ทำกิจกรรมของตน และหลับทันทีที่เข็มนาฬิกาเคลื่อนที่ออกไป) ดังนั้นแอพพลิเคชันแต่ละตัวจึงเป็นอิสระต่อกัน และเท่าเทียมกัน และเมื่อแอพพลิเคชันตัวใดตัวหนึ่งเกิดหยุดทำงาน (Not Responding) Process ตัวอื่นก็ยังคงได้รับวงรอบ CPU ที่ทำให้สามารถประมวลผลต่อไปได้ดังเดิม

ซึ่งแตกต่างกับระบบ Win16 (Non preemptive) เดิม ที่แอพพลิเคชันแต่ละตัวจะรันอยู่บนหน่วยความจำเดียวกัน (Single Address Space) คือเมื่อแอพพลิเคชันถูกปลุกให้ทำกิจกรรมของตน ระบบก็จะให้วงรอบ CPU ทั้งหมดกับแอพพลิเคชันนั้น และระบบจะได้วงรอบ CPU ทั้งหมดคืนก็ต่อเมื่อแอพพลิเคชันสิ้นสุดกิจกรรมนั้น ๆ แล้วระบบจึงส่งวงรอบ CPU นี้ให้กับแอพพลิเคชันอื่น ๆ ต่อไปดังที่กล่าวข้างต้น ปัญหาอยู่ที่ถ้าแอพพลิเคชันเกิดมีปัญหาขึ้น แล้วไม่คืนวงรอบ CPU ให้กับระบบนั่นหมายความว่าระบบได้สูญเสียการควบคุมอย่างสิ้นเชิง ทำให้ระบบและแอพพลิเคชันอื่น ๆ ไม่สามารถทำงานต่อไปได้ในทันที

Delphi & Multi-Threaded Applications

โดยปกติโปรแกรมที่สร้างขึ้นภายใต้ระบบ Win32 ด้วย Delphi 2.0 ขึ้นไปจะมี Threads 1 ตัวรันอยู่ภายใน Process เป็น Primary Threads มีความสำคัญคือคอยควบคุมการทำงานทั้งหมด ไม่ว่าจะเป็นการติดต่อกับวินโดวส์ แสดงผลข้อมูลผ่านทางชุด VCL Components ต่าง ๆ ของ Delphi เอง (เช่น Tpanel, Tedit ฯลฯ)

การสร้าง Thread Objects ขึ้นใหม่ภายใต้ Delphi ทำได้หลายวิธี วิธีแรกที่ขอกล่าวถึงคือใช้ฟังก์ชันของ Win32 เองด้วยคำสั่ง

```
Function CreateThread(
  lpThreadAttributes : Pointer;
  dwStackSize        : DWORD;
  lpStartAddress     : TFNThreadStartRoutine;
  lpParameter        : Pointer;
  dwCreationFlags    : DWORD;
  var  lpThreadID    : DWORD ) : Thandle; stdcall;
```

CreateThread เป็นคำสั่งที่ใช้สร้าง Thread Objects ภายใน Process เดียวกัน พารามิเตอร์แรก lpThreadAttributes ใช้บรรจุค่า Security Attributes (บนวินโดวส์ 95 ปกติกำหนดเป็น nil), dwStackSize ใช้ระบุขนาดของ Stack ที่ใช้ใน Threads นี้ (ถ้ากำหนดเป็น 0 จะใช้เท่ากับ Primary Thread), lpStartAddress ระบุตำแหน่ง Address ของ Thread Function (เป็นส่วนที่สำคัญที่สุด เพราะ Thread ใหม่ที่สร้างขึ้นจะทำงานตามคำสั่งที่ระบุอยู่ใน Thread Function นี้), lpParameter บรรจุค่าพารามิเตอร์ที่ต้องการผ่านให้กับ Thread Function, dwCreationFlags เป็นแฟล็กพิเศษ ใช้กำหนดการเริ่มทำงานของ Thread ถ้าระบุเป็น CREATE_SUSPENDED แล้วจะต้องเรียกฟังก์ชัน ResumeThread เพื่อให้ Thread Objects เริ่มทำงาน แต่ถ้าระบุเป็น 0 (NON-SUSPENDED) Thread ใหม่จะทำงานอัตโนมัติทันทีที่สร้าง Thread Objects สำเร็จ พารามิเตอร์ตัวสุดท้าย lpThreadID จะส่งค่าหมายเลข ID ของ Thread กลับมา และจะส่งค่า Handle ของ Thread Objects ที่สร้างสำเร็จนี้กลับมา เป็นค่า Thandle เพื่อใช้ในการอ้างถึงต่อไป (ใน Delphi 3.0 ขอให้เรียกใช้ BeginThread แทน CreateThread)

ตัวอย่าง Simple Threads

โปรแกรม uSimMain ดังลิสติงที่ 1 เป็นตัวอย่างแสดงการเรียกใช้ Thread Objects อย่างง่าย ๆ รวมทั้งแสดงให้เห็นข้อแตกต่างระหว่างโปรแกรมส่วนที่ใช้ Thread ประมวลผล และส่วนที่ไม่ใช้ Thread

การสร้างโปรแกรม uSimMain ให้สร้างโดยการเลือกเมนู New Application... เพื่อสร้าง Project ตัวใหม่ โดยให้เปลี่ยนชื่อฟอร์ม (Form Name) ในหน้าต่าง Object Inspector เป็น FsimMain และทำการ Save Project โดยการกดปุ่ม Save All... ให้ตั้งชื่อไฟล์ยูนิตว่า uSimMain และชื่อ Project เป็น SimThrd

สำหรับ Components ที่ใช้ให้ใส่ Tpanel Box 1 กรอบ และปุ่ม TBitBtn 2 ปุ่ม ดังรูปที่ 1 เพื่อใช้เป็นส่วนการแสดงผลและการเลือกการใช้งาน

ในยูนิต uSimMain ที่สร้างขึ้นจากหัวข้อข้างต้นให้เพิ่มโค้ดตามลิสติงที่ 1 (ที่แสดงเป็นตัวอักษรตัวหนาไว้) ดังนี้

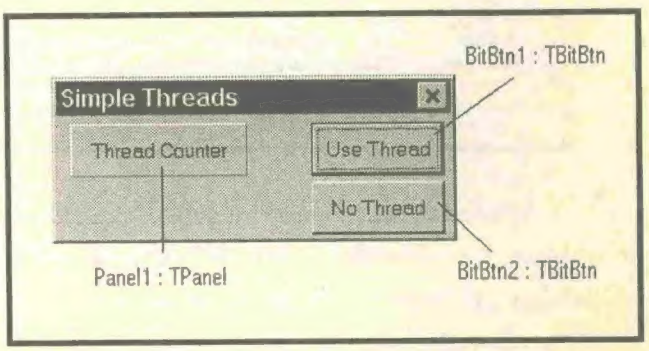
อธิบายโปรแกรม

บรรทัดที่ 6-7 ทำการสร้างเมสเสจ ID ใหม่ชื่อ WM_DISPLAYDATA เพื่อใช้อ้างถึงกิจกรรม WmDisplayData

บรรทัดที่ 18 กำหนดให้ตัวแปร hThread เป็นตัวแปรแบบ Thandle เพื่อใช้เก็บค่า Handle ของ Thread Objects ใหม่ เพื่อใช้ในการอ้างถึง Thread Objects เมื่อต้องการยกเลิกการใช้งาน

บรรทัดที่ 30-35 เป็นส่วนของฟังก์ชัน ThreadFunction จะถูกกำหนดไว้เป็น StdCall เพื่อให้การผ่านค่าพารามิเตอร์เป็นแบบ Standard Module (ความหมายและวิธีการใช้แบบอื่น ๆ ของ StdCall สามารถหาข้อมูลเพิ่มเติมได้ในระบบ Help ของ Delphi) โดยฟังก์ชันจะทำวงรอบ For..Next และส่งเมสเสจเป็นจำนวนรอบที่นับได้ให้กับกิจกรรม WmDisplayData ผ่านทางพารามิเตอร์ lparam เพื่อใช้ในการแสดงผล

บรรทัดที่ 37-41 เป็นส่วนของกิจกรรม WmDisplayData จะรับข้อมูลจากพารามิเตอร์ lparam แล้วนำไปแสดงผลออก Panel1 เพื่อแสดงการทำงานของ ThreadFunction



รูปที่ 1

ลิสติงที่ 1 โปรแกรม uSimMain

```
00001:  unit uSimMain;
00002:  interface
00003:  uses
00004:    Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
00005:    StdCtrls, Buttons, ExtCtrls;
00006:  const
00007:    WM_DISPLAYDATA = WM_USER + $100;
00008:  type
00009:    TFsimMain = class(TForm)
00010:      Panel1: TPanel;
00011:      BitBtn1: TBitBtn;
00012:      BitBtn2: TBitBtn;
00013:      procedure BitBtn1Click(Sender: TObject);
00014:      procedure BitBtn2Click(Sender: TObject);
00015:      procedure FormCloseQuery(Sender: TObject; var CanClose: Boolean);
00016:    private
00017:      { Private declarations }
00018:      hThread  : THandle;
00019:    public
00020:      { Public declarations }
00021:    protected
00022:      { Protected declarations }
00023:      procedure wmDisplayData( var mmsg : TMessage ); message WM_DISPLAYDATA;
00024:    end;
00025:  var
00026:    FsimMain: TFsimMain;
00027:  implementation
00028:  {$R *.DFM}
00029:  //- Thread Function ---------------------------------
00030:  function ThreadFunction : longint; stdCall;
00031:  var c : longInt;
00032:  begin
00033:    for c:= 1 to 10000 do
00034:      SendMessage( FsimMain.Handle, WM_DISPLAYDATA, 0, c );
00035:  end;
00036:  //- Messages Proc -----------------------------------
00037:  procedure TFsimMain.wmDisplayData( var mmsg : TMessage );
00038:  var s : string;
00039:  begin
00040:    s:=IntToStr( mmsg.lparam ); Panel1.caption:=s;
00041:  end;
00042:  //-------------------------------------------
00043:  procedure TFsimMain.BitBtn1Click(Sender: TObject);
00044:  var threadID : DWord;
00045:  begin // under Delphi 3.0 it's recommended that to use BeginThread instead.
00046:    hThread:=CreateThread( nil, 0, @ThreadFunction, nil, 0, threadID );
00047:    if ( hThread=0 ) then
00048:      ShowMessage( 'Can not create Thread Objects' );
00049:  end;
00050:  procedure TFsimMain.BitBtn2Click(Sender: TObject);
00051:  begin
00052:    ThreadFunction;
00053:  end;
00054:  procedure TFsimMain.FormCloseQuery(Sender: TObject; var CanClose: Boolean);
00055:  var exitCode : dword;
00056:  begin
00057:    if ( GetExitCodeThread( hThread, exitCode ) ) then EndThread( ExitCode );
00058:    CloseHandle( hThread ); canclose:=true;
00059:  end;
00060:  end.
```

บรรทัดที่ 46-48 เป็นส่วนของกิจกรรม BitBtn1Click เป็นการสร้าง Thread Objects ด้วยคำสั่ง CreateThread โดยผ่าน Address ของ ThreadFunction เพื่อให้ทำหน้าที่เป็น Thread Objects ใหม่

บรรทัดที่ 52 เป็นส่วนของกิจกรรม ทำการเรียกใช้ฟังก์ชัน ThreadFunction โดยไม่สร้างเป็น Thread Object

บรรทัดที่ 57-58 เป็นส่วนการยกเลิกการใช้ Thread Object โดยการเรียกฟังก์ชัน GetExitCodeThread เพื่อขอค่า ExitCode จากระบบ แล้วส่งให้โพรซีเดอร์ EndThread เพื่อยุติการรัน แล้วจึงยกเลิกหน่วยความจำ และรีซอร์สที่ใช้ใน Thread Object ด้วยคำสั่ง CloseHandle

การทำงานของโปรแกรมนี้คือเมื่อ User กดปุ่ม "Use Thread" ระบบจะทำการสร้าง Thread Objects ขึ้นใหม่เพื่อใช้ฟังก์ชัน ThreadFunction และถ้ากดปุ่ม "No Thread" แล้วตัวโปรแกรม (หรือ Primary Thread) จะเรียกใช้ฟังก์ชัน ThreadFunction เอง

ฟังก์ชัน ThreadFunction จะทำหน้าที่นับเลข และส่งตัวเลขที่นับแสดงผลออกที่ Panel1 โดยใช้กรรมวิธีการส่งเมสเสจ สาเหตุที่ใช้การส่งเมสเสจเนื่องจากฟังก์ชัน ThreadFunction ไม่ใช่กิจกรรมของ TFSimMain และการใช้คอมโปเนนต์ต่าง ๆ ใน TFSimMain จะต้องเป็นหน้าที่ของ Primary Thread เท่านั้น

เมื่อรันโปรแกรมนี้ จะพบข้อแตกต่างของการทำงานจากการกดปุ่มทั้งสองคือ

1. ถ้ารันด้วยปุ่ม "No Thread" : จะพบว่าในขณะที่โปรแกรมกำลังอยู่ในวนรอบ For..Next โปรแกรมจะไม่สามารถตอบสนองกับระบบได้ เช่น ไม่สามารถเลื่อนหน้าต่างได้

2. ถ้ารันด้วยปุ่ม "Use Thread" : ในขณะที่ Thread Objects ทำงานอยู่ในวงรอบ For..Next โปรแกรมยังคงสามารถตอบสนองกับระบบได้เป็นปกติ

Thread Priorities

เมื่อแอพพลิเคชันถูกรันขึ้น ระบบ Win32 จะสร้าง Process และ Thread Objects ขึ้น ในขณะเดียวกัน Process ก็จะกำหนดระดับความสำคัญ (Thread Priorities) ของ Thread Objects ด้วย Base Priorities Class

โดย Thread Objects แต่ละตัวจะมี Thread Priorities เป็นของตนเอง และระดับความสำคัญนี้จะมีผลต่อการดำเนินกิจกรรมของ Thread Objects แต่ละตัวบนระบบ เช่น ถ้ากำหนด Thread Priorities สำหรับ Thread Objects #1 ให้มีค่าเป็น THREAD_PRIORITY_HIGHEST แล้วระบบจะแบ่งวงรอบ CPU ให้กับ Thread Objects #1 มากกว่า Thread Objects อื่นที่มี Priorities ระดับต่ำกว่า (อาจเป็น THREAD_PRIORITY_NORMAL) ทำให้ Thread Objects #1 สามารถดำเนินกิจกรรมของตนเองมากกว่า Thread Objects อื่น เป็นต้น

ตารางที่ 1 Relative Priority Constants

Win32 Constant	Assigned Priority
THREAD_PRIORITY_ABOVE_NORMAL	Base priority + 1
THREAD_PRIORITY_BELOW_NORMAL	Base priority -1
THREAD_PRIORITY_HIGHEST	Base priority + 2
THREAD_PRIORITY_IDLE	ถ้า Base class เป็น Real-time Priority class = 16 ถ้าเป็น class อื่น = 1
THREAD_PRIORITY_LOWEST	Base priority -2
THREAD_PRIORITY_NORMAL	เท่ากับ Base priority
THREAD_PRIORITY_TIME_CRITICAL	ถ้า Base class เป็น Real-time Priority class = 31 ถ้าเป็น class อื่น = 15

เราสามารถปรับเปลี่ยนระดับของ Thread Priorities โดยการเรียกฟังก์ชันของ Win32 คือ

function SetThreadPriority(hThread : Thandle; nPriority : Integer) : BOOL; ฟังก์ชันสำหรับกำหนดค่า Thread Priorities ใหม่ โดย hThread คือค่า Handle ของ Thread Objects และ nPriority คือค่า Priority ที่ต้องการเปลี่ยน โดยค่านี้จะเปลี่ยนแปลงตามค่าของ Base Priorities Class ที่ใช้อยู่ตามตารางที่ 1 และจะให้ค่า True ถ้าสำเร็จ

function GetThreadPriority(hThread : Thandle) : integer; ฟังก์ชันสำหรับอ่านค่า Thread Priorities ที่ใช้อยู่ในปัจจุบัน โดย hThread คือค่า Handle ของ Thread Objects ที่ต้องการทราบ ถ้าสำเร็จ ฟังก์ชันนี้จะให้ค่าตามค่าในตารางที่ 1 ถ้าเกิด Error จะให้ค่า THREAD_PRIORITY_ERROR_RETURN

Synchronization Mechanisms

สำหรับระบบ Multi-Threaded Applications การควบคุมให้ Thread Objects ในระบบสามารถทำงานสอดคล้องกันอย่างสมบูรณ์เป็นสิ่งที่จำเป็นอย่างมาก ปัญหาและข้อผิดพลาดต่าง ๆ อาจเกิดขึ้นได้ถ้าหากการ Synchronize เป็นไปอย่างไม่ถูกต้อง เช่น ถ้า Application #1 มี Thread #A และ Thread #B ทำงานอยู่ในระบบและ Thread #A ต้องคำนวณผลลัพธ์ก่อน Thread #B นำไปคำนวณผลต่อ แล้ว Thread #B เกิดคำนวณผลก่อนที่ Thread #A จะดำเนินการคำนวณผล หรือยังคำนวณผลไม่เสร็จ นั่นคือ Thread #B ได้คำนวณผลผิดจากความเป็นจริง ทำให้ผลลัพธ์ที่ได้ไม่ถูกต้อง

ระบบ Synchronize Mechanisms บนระบบ Win32 สามารถทำได้หลายกรรมวิธี คือ Critical Sections, Events, Mutexes และ Semaphores ในที่นี้ผมขอกล่าวถึงเฉพาะ Critical Sections และ Mutexes เนื่องจากมีวิธีการใช้คล้ายคลึงกัน สำหรับ Evenst และ Semaphores จะนำมากล่าวถึงในโอกาสต่อไป

Critical Sections

Critical Sections เป็นกรรมวิธีที่ง่ายและเร็วที่สุดในการควบคุมการทำงานของ Thread ให้สามารถเข้าถึงข้อมูลได้อย่างเป็นระบบ ไม่คาบเกี่ยวกัน การใช้ระบบ Critical Sections จำเป็นที่จะต้องกำหนดเงื่อนไขการเข้าถึงข้อมูลให้กับทุก ๆ Thread Objects ที่ใช้อยู่ในแอพพลิเคชันให้เหมือนกัน คือ

1. ทำการล็อก (Lock) Critical Section Object ก่อนการเข้าถึงข้อมูลทุกครั้ง

2. เข้าถึงข้อมูล สามารถทำการ อ่าน เขียน หรือแก้ไขข้อมูลได้โดยปกติ

3. ทำการคลายล็อก (Unlock) Critical Section Object หลังจากเสร็จสิ้นการเข้าถึงข้อมูลทุกครั้งที่ล็อก

ชุดกิจกรรมในระบบ Critical Sections

Procedure InitializeCriticalSection(var CriticalSection : TRTLCriticalSection); เป็นกิจกรรมที่ใช้สำหรับ Initialized เข้าสู่ระบบ Critical Sections จะต้องเรียกใช้ทุกครั้งก่อนการใช้ระบบ Critical Sections

Procedure DeleteCriticalSection(var CriticalSection : TRTLCriticalSection); กิจกรรมสำหรับยกเลิกการใช้ระบบ Critical Sections

Procedure EnterCriticalSection(var CriticalSection : TRTLCriticalSection); กิจกรรมใช้สำหรับเข้าสู่สภาวะ Critical Sections Thread Objects ทุกตัวที่อยู่ในระบบจะต้องเรียกกิจกรรมนี้ก่อนการ access ตัวแปล Global

Procedure LeaveCriticalSection(var CriticalSection : TRTLCriticalSection); กิจกรรมใช้สำหรับออกจากสภาวะ Critical Sections จะเรียกใช้กิจกรรมนี้ หลังจาก EnterCriticalSection และเมื่อ access ตัวแปรแบบ Global เรียบร้อย ทำให้ Thread Objects อื่น ๆ สามารถเข้าสู่ระบบ Critical Sections ได้

ตัวอย่างการใช้งานระบบ Synchronize Mechanisms ด้วยระบบ Critical Section

● กำหนด CriticalSection Objects และ globalData ไว้ในส่วน Implementation (กรณีเขียนโปรแกรมแบบ Single Unit) โดย

```
implementation
{$R *.DFM}
var
  CriticalSection : TRTLCriticalSection;
  globalData      : longint;
```

● เพิ่มส่วน Initialized Critical Sections ไว้ในกิจกรรม CreateForm ของโปรแกรมหลัก

```
procedure TFcritcal.FormCreate(Sender: TObject);
begin
  InitializeCriticalSection( CriticalSection );
  Application.ProcessMessages;
  ...
end;
```

● เพิ่มส่วนยกเลิกการใช้ระบบ Critical Sections ไว้ในกิจกรรม FormCloseQuery เพื่อ Free resource ที่ใช้ในระบบ Critical Sections ทั้งหมด

```
procedure TFcritcal.FormCloseQuery(Sender:
TObject; var CanClose: Boolean);
begin
  DeleteCriticalSection( CriticalSection );
  CanClose:=true;
end;
```

● เพิ่มส่วนการเข้า-ออกสภาวะ Critical Sections ไว้ใน Thread Objects ทุกตัวเพื่อให้สามารถ Synchronize กับ Thread Objects อื่นได้

```
// Thread Objects Section ──────────────
function ThreadFunction1( p : pointer )
: Longint; stdCall;
begin
  EnterCriticalSection( CriticalSection );
// เข้าสู่ระบบ Critical Sections
  // ใส่ส่วนการ access ( Read, Write, Update )
ข้อมูลที่เป็น GlobalData ไว้ที่นี่
  // ...
  LeaveCriticalSection( CriticalSection );
// ออกจากระบบ Critical Sections
  result:=0;
end;
```

ข้อจำกัดของ Critical Sections คือ

1. สามารถควบคุม Thread Objects ที่ใช้งานอยู่ภายใน Process (Applications) เดียวกันเท่านั้น

2. ไม่มี Time Out ถ้า Thread Objects ที่ทำการล็อก Critical Section เกิดหยุดทำงาน แล้วไม่ได้ปลดล็อก Thread Objects อื่นจะไม่สามารถทำงานต่อไปได้

Mutexes

Mutexes (MUTual EXclusions) เป็นกรรมวิธีควบคุมระบบ Synchronize Mechanisms อีกแบบที่มีประสิทธิภาพมากกว่า Critical Sections (แต่ทำงานช้ากว่า) มีประสิทธิภาพโดยรวมดังนี้ คือ

1. สามารถควบคุม Thread Objects แบบหลาย Process ได้ กล่าวคือ สามารถใช้ควบคุม Thread Objects ที่อยู่ต่างโปรแกรม ให้ใช้ทรัพยากรร่วมกันอย่างมีระบบ

2. สามารถกำหนด Time Out ให้กับการ Synchronize ในระบบได้ เพื่อป้องกันการเกิด Dead lock

ชุดกิจกรรมที่ใช้ในระบบ Mutexes

Function CreateMutex(lpMutexAttributes : PSecurity Attributes; bInitialOwner : BOOL; lpName : Pchar) : Thandle; stdcall; เป็นกิจกรรมสำหรับ Initialized เข้าสู่ระบบ MUTEXES มี lpMutexAttributes ผ่านค่า security attributes ให้กับระบบ Mutex ถ้ากำหนดให้เป็น nil จะใช้ default security attributes ของระบบ, bInitialOwner ระบุการเป็นเจ้าของระบบ Mutex ถ้ากำหนดเป็น TRUE แล้วเมื่อ Initialized สำหรับจะเข้าสู่ระบบ Mutex ทันที (Signaled), lpName ระบุชื่อ Mutex Name ใช้ในกรณี OpenMutex ใน Process อื่น ถ้า Initialized สำเร็จจะ Return ค่าเป็น Handle ของ Mutex ที่สร้างขึ้นใหม่

Function WaitForSingleObject(hHandle : Thandle; dwMilliseconds : DWORD) : DWORD; กิจกรรมสำหรับเข้าสู่ Mutex State โดย hHandle ระบุ Handle ของ Mutex ที่ใช้, dwMiiliseconds กำหนดเวลาสูงสุดที่รอให้ Thread Objects ทำกิจกรรมเสร็จ (หน่วยเป็น Milliseconds) สามารถกำหนดเป็นค่า INFINITE เพื่อให้รอแบบไม่สิ้นสุด สำหรับกำหนดให้ระบบรอจนกว่า Thread Objects ที่ครองระบบอยู่ทำงานจนเสร็จสิ้น

Function ReleaseMUTEX(hHandle : Thandle) : BOOL; กิจกรรมใช้สำหรับออกจาก Mutex State เพื่อคืน Resources ให้กับระบบ ทำให้ Thread Objects อื่น ๆ สามารถเข้าสู่ระบบ Mutex ได้

Function CloseHandle(hHandle : Thandle) : BOOL; กิจกรรมใช้สำหรับยกเลิกค่า handle จะใช้เมื่อต้องการยกเลิกการใช้ระบบ Mutex

วิธีการใช้งานระบบ Synchronize Mechanisms ด้วยระบบ Mutex

การใช้งานระบบ Mutex จะคล้ายกับระบบ Critical Sections กล่าวคือ

- กำหนด hMutex เป็น Thandle ไว้ในส่วน Global Variables

```
var
      hMutex      : THandle;
```

- Initialized ระบบ Mutex ในกิจกรรม FormCreate

```
procedure TFmutexes.FormCreate(Sender: TObject);
begin
  hMutex:=CreateMutex( nil, False, nil );
end;
```

- เพิ่มส่วนการเข้า-ออกระบบ Mutex ไว้ใน Thread Objects ทุกตัวเพื่อให้สามารถ Synchronize กับ Thread Objects อื่นได้

```
function ThreadFunction1(p:pointer):
Longint;stdCall;
begin
  WaitForSingleObject(hMutex,INFINITE);
// เข้าสู่ระบบ Mutex
// ใส่ส่วนการ access (Read,Write,Update) ข้อมูลที่เป็น
GlobalData ไว้ที่นี่
// ...
  ReleaseMutex(hMutex);// ออกจากระบบ Mutex
  result:=0;
end;
```

- เพิ่มส่วนยกเลิกการใช้ระบบ Mutex ไว้ในกิจกรรม FormCloseQuery เพื่อ Free resource ที่ใช้ทั้งหมด

```
procedure TFmutexes.FormCloseQuery(Sender:
TObject; var CanClose: Boolean);
begin
  CloseHandle( hMutex ); canclose:=true;
end;
```

จากคำอธิบายและตัวอย่างดังกล่าวข้างต้นทั้งหมด เป็นการใช้คำของระบบ Win32 ในการสร้างระบบ Multi-Threaded Applications ซึ่งอาจจะต้องใช้เวลาในการทำความเข้าใจการทำงานของ Thread Objects รวมทั้งการเลือกระบบ Synchronize Mechanisms ที่เหมาะสมมาใช้กับ Thread Objects ที่สร้างขึ้น ยังมีอีกวิธีการในการสร้าง Thread Objects ได้ง่ายและรวดเร็วกว่าคือ ใช้ Delphi's TThread Class

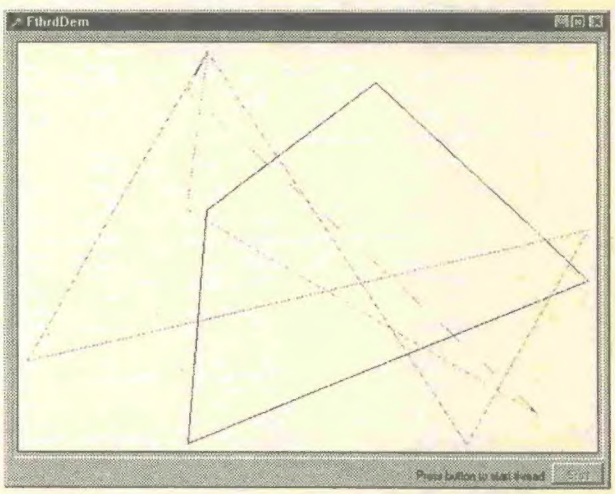
Delphi's TThread Class

TThread Class เป็น Class ใหม่ของ Delphi มีอยู่ในรุ่น 2.0 และ 3.0 มีลักษณะเป็น Virtual และ Abstract Class ที่ผู้ใช้ไม่สามารถเรียกใช้กิจกรรมที่สร้างได้โดยตรง แต่จะต้องเรียกใช้ผ่านกิจกรรม Execute ที่กำหนดให้เท่านั้น เราสามารถสร้าง TThread Objects ได้โดย

- เลือกเมนู File แล้วเลือก New...
- เลือก Thread Objects ใน New Pages (ในหน้าต่าง New Items)
- ตั้งชื่อ Class Name ให้กับ Thread Objects ตัวใหม่
- โปรแกรม Delphi จะทำการสร้าง Thread Objects ใหม่ให้กับระบบ

การเขียนโปรแกรม Thread Objects ขอให้ศึกษาจากโปรเจ็กต์ ThrdDemo.Dpr โดยโปรแกรม ThrdDemo จะแสดงการทำงานที่เป็นอิสระต่อกันของ Thread Objects 3 ตัว แสดงอยู่ในรูปแบบ Spin Line สีต่าง ๆ วิ่งอยู่ในหน้าต่างหลัก ซึ่งแต่ละสีใช้แทน Thread Objects 1 ตัว

โปรแกรม ThrdDemo จะแบ่งออกเป็น 2 ส่วนคือ ส่วนของโปรแกรมหลัก แสดงอยู่ในลิสติงที่ 2 และส่วนที่เป็น Thread Objects แสดงอยู่ในลิสติงที่ 3 สำหรับคอมโปเนนต์และค่าต่าง ๆ ที่ใช้แสดงอยู่ในลิสติงที่ 4



รูปที่ 2 โปรแกรม ThrdDemo

ลิสติงที่ 2 โปรแกรม uThrdDem.pas

```
00001:  unit uthrdDem;
00002:  interface
00003:  uses
00004:    Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
00005:    ExtCtrls, StdCtrls, Buttons, uSpnThrd;
00006:  type
00007:    TFthrdDem = class(TForm)
00008:      BitBtn1: TBitBtn;
00009:      Bevel1: TBevel;
00010:      Panel1: TPanel;
00011:      PaintBox: TPaintBox;
00012:      Label1: TLabel;
00013:      procedure BitBtn1Click(Sender: TObject);
00014:    private
00015:      { Private declarations }
00016:        threadInfo : TthreadInfoR;
00017:    public
00018:      { Public declarations }
00019:    end;
00020:  var
00021:    FthrdDem: TFthrdDem;
00022:  implementation
00023:  {$R *.DFM}
00024:  procedure TFthrdDem.BitBtn1Click(Sender: TObject);
00025:  begin
00026:    with threadInfo, PaintBox do
00027:    begin _left:=left; _top:=top; _width:=width; _height:=height; end;
00028:    // Create 1st Thread
00029:    threadInfo._color:=clLime; TSpinLine.Create( PaintBox, threadInfo );
00030:    // Create 2nd Thread
00031:    threadInfo._color:=clAqua; TSpinLine.Create( PaintBox, threadInfo );
00032:    // Create 3rd Thread
00033:    threadInfo._color:=clRed;  TSpinLine.Create( PaintBox, threadInfo );
00034:    BitBtn1.enabled:=false;
00035:  end;
00036:  end.
```

ลิสติงที่ 3 โปรแกรม uSpnThrd.pas

```
00001:  unit uSpnThrd;
00002:  interface
00003:  uses Classes, Graphics, Windows, extctrls;
00004:  const
00005:    MOVE_STEP   = 10;
00006:    MOVE_UP     = 1;  MOVE_DOWN   = 2;  MOVE_RIGHT  = 3;  MOVE_LEFT   = 4;
00007:    MOVE_UR     = 5;  MOVE_DR     = 6;  MOVE_UL     = 7;  MOVE_DL     = 8;
00008:  type
00009:    TthreadInfoR = record
00010:      _left, _top, _width, _height, _color : integer;
00011:    end;
00012:    TspinLineR = record
00013:      lineColor : TColor;
00014:      p1, p2, p3, p4      : TPoint;
00015:      sp1, sp2, sp3, sp4 : Tpoint;
00016:      dp1, dp2, dp3, dp4 : integer;
00017:    end;
00018:    TSpinLine = class(TThread)
00019:    public
00020:      { Public declarations }
00021:      constructor Create( Pbox: TPaintBox; tinfo : TthreadInfoR );
00022:    private
00023:      { Private declarations }
00024:      paintBox : TpaintBox;
```

→

ลิสติงที่ 8 (ต่อ)

```
00025:       spineLine : TspinLineR;
00026:       cor_left, cor_right, cor_top, cor_bottom : integer;
00027:       procedure InitVar;
00028:       procedure RandomPoint( var p : TPoint );
00029:       procedure RandomValue;
00030:       procedure DoMovePosition( var d : integer; var p : Tpoint );
00031:       procedure DoCalPosition;
00032:       procedure DoPaintSPLine;
00033:       procedure DoSpinLineThread;
00034:     protected
00035:       procedure Execute; override;
00036:     end;
00037:   implementation
00038:   //- Spine Routine ------------------------------
00039:   procedure TSpinLine.InitVar;
00040:   begin
00041:     fillchar( spineLine, sizeof( TspinLineR ), #0 );
00042:   end;
00043:   procedure TSpinLine.RandomPoint( var p : TPoint );
00044:   begin
00045:     Randomize;
00046:     with P do begin x:=random( PaintBox.Width ); y:=random( PaintBox.Height ); end;
00047:   end;
00048:   procedure TSpinLine.RandomValue;
00049:   begin
00050:     with spineLine do
00051:     begin
00052:       lineColor:=clLime;
00053:       RanDomPoint( p1 ); dp1:=random( 9 );
00054:       RanDomPoint( p2 ); dp2:=random( 9 );
00055:       RanDomPoint( p3 ); dp3:=random( 9 );
00056:       RanDomPoint( p4 ); dp4:=random( 9 );
00057:     end;
00058:   end;
00059:   procedure TSpinLine.DoMovePosition( var d : integer; var p : Tpoint );
00060:   begin
00061:     case d of
00062:       MOVE_UP :
00063:       begin
00064:         dec( p.y,MOVE_STEP ); if ( p.y<=cor_top ) then begin d:=random( 9 );
p.y:=cor_top; end;
00065:       end;
00066:       MOVE_DOWN :
00067:       begin
00068:         inc( p.y,MOVE_STEP ); if ( p.y>=cor_bottom ) then begin d:=random( 9 );
p.y:=cor_bottom; end;
00069:       end;
00070:       MOVE_RIGHT :
00071:       begin
00072:         inc( p.x,MOVE_STEP ); if ( p.x>=cor_Right ) then begin d:=random( 9 );
p.x:=cor_Right; end;
00073:       end;
00074:       MOVE_LEFT :
00075:       begin
00076:         dec( p.x,MOVE_STEP ); if ( p.x<=cor_Left ) then begin d:=random( 9 );
p.x:=cor_Left; end;
00077:       end;
00078:       MOVE_UR :
00079:       begin
00080:         dec( p.y,MOVE_STEP ); if ( p.y<=cor_top ) then begin d:=random( 9 );
p.y:=cor_top; end;
00081:         inc( p.x,MOVE_STEP ); if ( p.x>=cor_Right ) then begin d:=random( 9 );
p.x:=cor_Right; end;
00082:       end;
00083:       MOVE_DR :
00084:       begin
00085:         inc( p.y,MOVE_STEP ); if ( p.y>=cor_bottom ) then begin d:=random( 9 );
p.y:=cor_bottom; end;
00086:         inc( p.x,MOVE_STEP ); if ( p.x>=cor_Right ) then begin d:=random( 9 );
```

→

ลิสติงที่ 3 (ต่อ)

```
p.x:=cor_Right; end;
00087:       end;
00088:       MOVE_UL :
00089:       begin
00090:         dec( p.y,MOVE_STEP ); if ( p.y<=cor_top ) then begin d:=random( 9 );
p.y:=cor_top; end;
00091:         dec( p.x,MOVE_STEP ); if ( p.x<=cor_Left ) then begin d:=random( 9 );
p.x:=cor_Left; end;
00092:       end;
00093:       MOVE_DL :
00094:       begin
00095:         inc( p.y,MOVE_STEP ); if ( p.y>=cor_bottom ) then begin d:=random( 9 );
p.y:=cor_bottom; end;
00096:         dec( p.x,MOVE_STEP ); if ( p.x<=cor_Left ) then begin d:=random( 9 );
p.x:=cor_Left; end;
00097:       end;
00098:       else d:=random( 9 );
00099:     end;
00100:   end;
00101:   procedure TSpinLine.DoCalPosition;
00102:   begin
00103:     with spineLine do
00104:     begin
00105:       sp1:=p1; DoMovePosition( dp1, p1 ); sp2:=p2; DoMovePosition( dp2, p2 );
00106:       sp3:=p3; DoMovePosition( dp3, p3 ); sp4:=p4; DoMovePosition( dp4, p4 );
00107:     end;
00108:   end;
00109:   procedure TSpinLine.DoPaintSPLine;
00110:   var lastColor : TColor;
00111:   begin
00112:     with PaintBox.canvas, spineLine do
00113:     begin
00114:       Pen.Color:=clBlack;
00115:       MoveTo( sp1.x, sp1.y ); LineTo( sp2.x, sp2.y );
00116:       MoveTo( sp2.x, sp2.y ); LineTo( sp3.x, sp3.y );
00117:       MoveTo( sp3.x, sp3.y ); LineTo( sp4.x, sp4.y );
00118:       MoveTo( sp4.x, sp4.y ); LineTo( sp1.x, sp1.y );
00119:       Pen.Color:=lineColor;
00120:       MoveTo( p1.x, p1.y ); LineTo( p2.x, p2.y );
00121:       MoveTo( p2.x, p2.y ); LineTo( p3.x, p3.y );
00122:       MoveTo( p3.x, p3.y ); LineTo( p4.x, p4.y );
00123:       MoveTo( p4.x, p4.y ); LineTo( p1.x, p1.y );
00124:     end;
00125:   end;
00126:   //-----------------------------------------
00127:   constructor TSpinLine.Create( Pbox: TPaintBox; tinfo : TthreadInfoR );
00128:   begin
00129:     paintBox:=Pbox; InitVar; RandomValue;
00130:     with tinfo do
00131:     begin
00132:       cor_left:=_left; cor_right:=_width-_left; cor_top:=_top; cor_bottom:=_height-_top;
00133:       spineLine.lineColor:=_color;
00134:     end;
00135:     FreeOnTerminate := True;
00136:     inherited Create(False);
00137:   end;
00138:   procedure TSpinLine.Execute;
00139:   begin
00140:     DoSpinLineThread;
00141:   end;
00142:   procedure TSpinLine.DoSpinLineThread;
00143:   begin
00144:     repeat
00145:       DoCalPosition;
00146:       Synchronize( DoPaintSPLine );
00147:       sleep(20);
00148:     until ( Terminated );
00149:   end;
00150:   end.
```

■

ลิสติงที่ 4 คอมโปเนนต์และค่าต่าง ๆ ที่ใช้

```
object FthrdDem: TFthrdDem
  ClientHeight = 453
  ClientWidth = 600
  object Bevel1: TBevel
    Left = 5
    Top = 5
    Width = 591
    Height = 421
  end
  object BitBtn1: TBitBtn
    Left = 545
    Top = 430
    Width = 51
    Height = 21
    Caption = 'Start'
    OnClick = BitBtn1Click
  end
  object Panel1: TPanel
    Left = 10
    Top = 10
    Width = 581
    Height = 411
    Color = clBlack
    object PaintBox: TPaintBox
      Left = 5
      Top = 5
      Width = 571
      Height = 401
    end
  end
end
```

อธิบายโปรแกรม

ในส่วนของโปรแกรมหลัก (ลิสติงที่ 2) ผมขอกล่าวถึงเฉพาะกิจกรรม BitBtn1Click เป็นกิจกรรมสำหรับสร้าง Thread Objects ด้วยคำสั่ง TSpinLine.Create(PaintBox, threadInfo); โดยส่ง Paintbox Objects เพื่อใช้สำหรับเป็นพื้นที่ในการแสดงผล และ threadInfo เก็บค่าความกว้างของ PaintBox และสีที่ใช้สำหรับแสดงผล (จุดนี้ต้องการแสดงให้เห็นว่าเราสามารถผ่านข้อมูลให้กับ Thread Objects แต่ละตัวที่สร้างขึ้นเหมือนกับ Thread Objects ที่สร้างด้วยคำสั่งของ Win32)

สำหรับส่วนของโปรแกรม Thread Objects (ลิสติงที่ 3) ขอกล่าวถึงเฉพาะส่วนที่สำคัญดังต่อไปนี้

บรรทัดที่ 5-7 กำหนดค่า Constant ที่ใช้ในระบบ ใช้สำหรับกำหนดทิศทางการเคลื่อนที่ทั้ง 8 ทิศ

บรรทัดที่ 9-17 กำหนดรูปแบบข้อมูลคือ TthreadInfoR เป็น Structure ที่ใช้ส่งข้อมูลให้กับ Thread Objects และ TspinLineR เป็น Structure ที่ใช้ดำเนินกิจกรรมในระบบ

บรรทัดที่ 43-58 กิจกรรม RandomPoint และกิจกรรม RandomValue; ใช้สำหรับสุ่มตำแหน่งของ SpineLine เพื่อใช้ระบุตำแหน่งเริ่มแรก

บรรทัดที่ 59-108 กิจกรรม DoMovePosition และกิจกรรม DoCalPosition; ใช้คำนวณหาทิศทางการเคลื่อนที่ โดยเมื่อเคลื่อนที่ไปถึงขอบ PaintBox แล้วให้สุ่มทิศทางการเคลื่อนที่ใหม่

บรรทัดที่ 109-125 กิจกรรม DoPaintSPLine; เป็นกิจกรรมวาดรูป SpineLine การใช้งานจะต้องเรียกผ่านกิจกรรม Synchronize(Method: TThreadMethod); ของ Thread Objects (การเรียกใช้งาน VCL Components และข้อมูลชนิด Global ใน Thread Objects จะต้องเรียกผ่านกิจกรรม Synchronize เท่านั้น)

บรรทัดที่ 129-134 เป็นส่วนการคัดลอกข้อมูลจากตัวแปรที่ส่งมาให้

บรรทัดที่ 135 กำหนด FreeOnTerminate:=True; เป็นการระบุให้ Thread Objects ยกเลิกการใช้งานด้วยตัวเองถ้าปิดแอพพลิเคชัน (Primary Thread) โดยการเซตให้ Terminated Flag มีค่าเท่ากับ TRUE

บรรทัดที่ 136 inherited Create(False); เป็นการเรียกกิจกรรม Create จาก Objects ต้นสกุล โดยผ่านพารามิเตอร์ FALSE ซึ่งหมายถึงให้ทำงานทันทีเมื่อรันโปรแกรม Thread Objects

บรรทัดที่ 138-141 กิจกรรม Execute; เป็นหัวใจหลักของการทำงานในระบบ Thread Objects (เปรียบเสมือนเป็น Main Program ของระบบ Thread) คือกิจกรรมทั้งหมดที่สร้างขึ้นจะสามารถเรียกใช้งานได้ต้องเรียกผ่านภายใต้กิจกรรม Execute เท่านั้น (ทำการ overridden โดยการเขียนกิจกรรมนี้ขึ้นใหม่ให้เรียกกิจกรรมอื่น ๆ ที่สร้างขึ้น)

บรรทัดที่ 146 เป็นการเรียกใช้กิจกรรม Synchronize

บรรรทัดที่ 147 เรียกกิจกรรม Sleep(Milliseconds : DWORD); เพื่อหน่วยเวลา Thread Objects ไว้เท่ากับ Milliseconds

บรรทัดที่ 144-148 เป็นวนรอบการทำงานของระบบ Thread Objects นี้ (สังเกตจะเรียกผ่านกิจกรรม Execute) โดยจะหยุดทำงานเมื่อ Terminated Flag เป็น TRUE

จากตัวอย่างโปรแกรมข้างต้นเป็นเพียงแนวความคิดที่ผมเขียนให้เห็นการทำงานของระบบ Thread Objects และเพื่อที่จะให้ยังคงอยู่ในรูปแบบการเขียนโปรแกรมแบบ Objects มากที่สุด จึงใช้ Delphi's TThread Objects Class เป็นตัวอย่างหลัก ซึ่งน่าจะใช้งานได้ง่ายมากกว่าการเขียนด้วยแบบอื่น สวัสดีครับ